3.11 Working with Multiple Quantifiers

Expressions in predicate logic with a single quantifier can generally be translated into English as either "there exists an element x of set S that satisfies $P(x)$" (existential quantifier) or "every element x of set S satisfies $P(x)$" (universal quantifier). However, there are situations where multiple variables are quantified, and we need to pay special attention to what such statements are actually saying. Let us revisit our $Loves$ predicate from earlier this chapter. In particular, recall the following relationships regarding who loves whom:

	Sophia	**Thelonious**	**Stanley**	**Laura**
Breanna	False	True	True	False
Malena	False	True	True	True
Patrick	False	False	True	False
Ella	False	False	True	True

Remember that our $Loves$ predicate is binary—what if we wanted to quantify *both* of its inputs? Consider the formula:

$$\forall a \in A,\ \forall b \in B,\ Loves(a, b).$$

We translate this as "for every person a in A, for every person b in B, a loves b." After some thought, we notice that the order in which we quantified a and b doesn't matter; the statement "for every person b in B, for every person a in A, a loves b" means exactly the same thing! In both cases, we are considering all possible pairs of people (one from A and one from B).

In general, when we have two consecutive universal quantifiers the order does *not* matter. That is, the following two formulas are equivalent:[1]

[1] Tip: when the domains of the two variables are the same, we typically combine the quantifications, e.g., $\forall x \in S,\ \forall y \in S,\ P(x, y)$ into $\forall x, y \in S,\ P(x, y)$.

- $\forall x \in S_1,\ \forall y \in S_2,\ P(x, y)$
- $\forall y \in S_2,\ \forall x \in S_1,\ P(x, y)$

The same is true of two consecutive existential quantifiers. Consider the statements "there exist an a in A and b in B such that a loves b" and "there exist a b in B and a in A such that a loves b." Again, they mean the same thing: in this case, we only care about one particular pair of people (one from A and one from B), so the order in which we pick the particular a and b doesn't matter. In general, the following two formulas are equivalent:

- $\exists x \in S_1,\ \exists y \in S_2,\ P(x, y)$

- $\exists y \in S_2,\ \exists x \in S_1,\ P(x, y)$

But even though consecutive quantifiers of the same type behave very nicely, this is **not** the case for a pair of alternating quantifiers. First, consider

$$\forall a \in A,\ \exists b \in B,\ Loves(a, b).$$

This can be translated as "For every person a in A, there exists a person b in B, such that a loves b."[2] This is true: every person in A loves at least one person.

[2] Or put a bit more naturally, "For every person a in A, a loves someone in B," which can be shortened even further to "Everyone in A loves someone in B."

a **(from** A**)**	b **(a person in** B **who** a **loves)**
Breanna	Thelonious
Malena	Laura
Patrick	Stanley
Ella	Stanley

Note that the choice of person who a loves depends on a: this is consistent with the latter part of the English translation, "a loves someone in B."

Let us contrast this with the similar-looking formula, where the order of the quantifiers has changed:

$$\exists b \in B,\ \forall a \in A,\ Loves(a, b).$$

This formula's meaning is quite different: "there exists a person b in B, where for every person a in A, a loves b." Put more naturally, "there is a person b in B who is loved by everyone in A" or "someone in B is loved by everyone in A".

b **(from** B**)**	**Loved by everyone in** A**?**
Sophia	No
Thelonious	No
Stanley	Yes
Laura	No

This happens to be True because everyone in A loves Stanley. But it would *not* be True if we, for example, removed the love connection between Malena and Stanley. In this case, Stanley would no longer be loved by everyone, and so *no one* in B is loved by everyone in A. But notice that even if Malena no longer loves Stanley, the previous statement ("everyone in A loves someone") remains True!

So we would have a case where switching the order of quantifiers changes the meaning of a formula! In both cases, the existential quantifier $\exists b \in B$ involves making a *choice* of person from B. But in the first case, this quantifier occurs after a is quantified, so the choice

of b is allowed to depend on the choice of a. In the second case, this quantifier occurs before a, and so the choice of b must be *independent* of the choice of a.

When reading a nested quantified expression, you should read it from left to right, and pay attention to the order of the quantifiers. In order to see if the statement is True, whenever you come across a universal quantifier, you must verify the statement for every single value that this variable can take on. Whenever you see an existential quantifier, you only need to exhibit *one* value for that variable such that the statement is True, and this value can depend on the variables to the left of it, but not on the variables to the right of it.

Translating multiple quantifiers into Python code

Now let's see how we could represent this example in Python. First, recall the table of who loves whom from above:

	Sophia	**Thelonious**	**Stanley**	**Laura**
Breanna	False	True	True	False
Malena	False	True	True	True
Patrick	False	False	True	False
Ella	False	False	True	True

And we can represent this table of who loves whom in Python as a *list of lists* or, more precisely, using a `list[list[bool]]`.

```
[
  [False, True, True, False],
  [False, True, True, True],
  [False, False, True, False],
  [False, False, True, True]
]
```

Our list is the same as the table above, except with the people's names removed. Each *row* of the table represents a person from set A, while each *column* in the table represents a person from set B. We've kept the order the same; so the first row represents $Breanna$, while the third column represents $Stanley$.

Now, how are we going to access the data from this table? For this section we're going to put all of our work into a new file called `loves.py`, and so we'll start by defining a new variable in this file:

```
# In loves.py
LOVES_TABLE = [
  [False, True, True, False],
  [False, True, True, True],
  [False, False, True, False],
```

```
    [False, False, True, True]
]
```

This is the first time we've defined a variable within a Python file (rather than the Python console) that is *not* in a function definition. Variables defined in this way are called *global constants*, to distinguish them from the local variables defined within functions.[3] Global

[3] The term "constant" is not important right now, but will become important later in the course.

constants are called "global" because their *scope* is the entire Python module in which they are defined: they can be accessed anywhere in the file, including all function bodies. They can also be imported and used in other Python modules, and are available when we run the file in the Python console.

Exploring `LOVES_TABLE`

To start, let's run our `loves.py` file in the Python console so we can play around with the `LOVES_TABLE` value. Because `LOVES_TABLE` is a list of lists, where each inner list represents a row of the table, it's easy to access a single row with list indexing:

```
>>> LOVES_TABLE[0]  # This is the first row of the table
[False, True, True, False]
```

From here, we can access individual elements of the table, which represent an individual value of the $Loves(a, b)$ predicate.

```
>>> LOVES_TABLE[0][1]  # This is the (0, 1) entry in the table
True
>>> LOVES_TABLE[2][3]  # This is the (2, 3) entry in the table
False
```

In general, `LOVES_TABLE[i][j]` evaluates to the entry in row `i` and column `j` of the table. Finally, since the data is stored by rows, accessing columns is a little more work. To access column `j`, we can use a list comprehension to access the `j`-th element in each row:

```
>>> [LOVES_TABLE[i][0] for i in range(0, 4)]
[False, False, False, False]
```

Now, let's return to our Python file `loves.py` and define a version of our $Loves$ predicate. First, we add two more constants to represent the sets A and B, but using a dictionary to map names to their corresponding indices in `LOVES_TABLE`.

```
# In loves.py
LOVES_TABLE = [
    [False, True, True, False],
    [False, True, True, True],
    [False, False, True, False],
    [False, False, True, True]
]

A = {
    'Breanna': 0,
    'Malena': 1,
    'Patrick': 2,
    'Ella': 3
}

B = {
    'Sophia': 0,
    'Thelonius': 1,
    'Stanley': 2,
    'Laura': 3,
}
```

Next, we define a `loves` predicate, which take in two strings (note the preconditions) and returns whether person `a` loves person `b`.[4]

[4] Note that because this function is defined in the same file as `LOVES_TABLE`, it can access that global constant in its body.

```
def loves(a: str, b: str) -> bool:
    """Return whether the person at index a loves the person at index b.

    Preconditions:
      - a in A
      - b in B

    >>> loves('Breanna', 'Sophia')
    False
    """
    a_index = A[a]
    b_index = B[b]
    return LOVES_TABLE[a_index][b_index]
```

Now that we've seen how to access individual entries, rows, and columns from the table, let's turn to how we would represent the statements in predicate logic we've written in this section. First, we can express $\forall a \in A,\ \forall b \in B,\ Loves(a, b)$ as the expression:

```
>>> all({loves(a, b) for a in A for b in B})
False
```

And similarly, we can express $\exists a \in A,\ \exists b \in B,\ Loves(a, b)$ as the expression:

```
>>> any({loves(a, b) for a in A for b in B})
True
```

These two examples illustrate how Python's `all` and `any` functions naturally enable us to express multiple quantifiers of the same type. But what about the expressions we looked at with alternating quantifiers? Consider $\forall a \in A,\ \exists b \in B,\ Loves(a, b)$. It is possible to construct a nested expression that represents this one as well:

```
>>> all({any({loves(a, b) for b in B}) for a in A})
True
```

Though this is structurally equivalent to the statement in predicate logic, it's syntactically longer and a bit harder to read. In general we try to avoid lots of nesting in expressions in programming, and a rule of thumb we'll try to follow in this course is to *never nest `all`/`any` calls*. Instead, we can pull out the inner `any` into its own function, which not only reduces the nesting but makes it clearer what's going on:

```
def loves_someone(a: str) -> bool:
    """Return whether a loves at least one person in B.

    Preconditions:
      - a in A
    """
    return any({loves(a, b) for b in B})


>>> all({loves_someone(a) for a in A})
True
```

Similarly, we can express the statement $\exists b \in B,\ \forall a \in A,\ Loves(a, b)$ in two different ways. With a nested `any`/`all`:

```
>>> any({all({loves(a, b)} for a in A) for b in B})
True
```

And by pulling out the inner `all` expression into a named function:

```
def loved_by_everyone(b: str) -> bool:
    """Return whether b is loved by everyone in A.
```

```
    Preconditions:
      - b in B
    """
    return all({loves(a, b)} for a in A)


>>> any({loved_by_everyone(b) for b in B})
True
```

Here is our final `loves.py` file, for you to play around with:

```
# In loves.py
LOVES_TABLE = [
    [False, True, True, False],
    [False, True, True, True],
    [False, False, True, False],
    [False, False, True, True]
]

A = {
    'Breanna': 0,
    'Malena': 1,
    'Patrick': 2,
    'Ella': 3
}

B = {
    'Sophia': 0,
    'Thelonius': 1,
    'Stanley': 2,
    'Laura': 3,
}

def loves(a: str, b: str) -> bool:
    """Return whether the person at index a loves the person at index b.

    Preconditions:
      - a in A
      - b in B

    >>> loves('Breanna', 'Sophia')
    False
    """
    a_index = A[a]
    b_index = B[b]
    return LOVES_TABLE[a_index][b_index]


def loves_someone(a: str) -> bool:
    """Return whether a loves at least one person in B.

    Preconditions:
```

```
        - a in A
    """
    return any({loves(a, b) for b in B})


def loved_by_everyone(b: str) -> bool:
    """Return whether b is loved by everyone in A.

    Preconditions:
        - b in B
    """
    return all({loves(a, b)} for a in A)
```

A famous logical statement

Before we wrap up, let us use our understanding of multiple quantifiers to express one of the more famous properties about prime numbers: "there are infinitely many primes."[5]

[5] Later on, we'll actually prove this statement!

In Section 2.9, we saw how to express the fact that a single number p is a prime number, but how do we capture "infinitely many"? The key idea is that because primes are natural numbers, if there are infinitely many of them, then they have to keep growing bigger and bigger.[6] So we can express the original statement as "every natural number has a prime

[6] Another way to think about this is to consider the statement "every prime number is less than 9000. If this statement were True, then there could only be at most 8999 primes."

number larger than it," or in the symbolic notation:

$$\forall n \in \mathbb{N},\ \exists p \in \mathbb{N},\ p > n \wedge IsPrime(p).$$

If we wanted to express this statement without either the $IsPrime$ or divisibility predicates, we would end up with an extremely cumbersome statement:

$$\forall n \in \mathbb{N},\ \exists p \in \mathbb{N},\ p > n \wedge p > 1 \wedge \Big(\forall d \in \mathbb{N},\ \ (\exists k \in \mathbb{Z},\ p = kd) \Rightarrow d = 1 \vee d = p\Big).$$

This statement is terribly ugly, which is why we define our own predicates! Keep this in mind throughout the course: when you are given a statement to express, make sure you are aware of all of the relevant definitions, and make use of them to simplify your expression.

Looking Ahead

In this section, we've introduced the notion of lists within lists to represent tables of values for binary predicates. In the next chapter, we'll start looking at tabular data and other forms of nested collections of data in more detail, and see how these more complex structures can be used to represent real-world data for our programs.